基于圈复杂度的阶段动态符号执行*

毕雪洁[1],於家伟[1],李世明[1,2]

(1.哈尔滨师范大学 计算机科学与信息工程学院,黑龙江 哈尔滨 150025;
2.上海市信息安全综合管理技术研究重点实验室,上海 200240)

摘 要:为了缓解动态符号执行不可避免的路径爆炸等问题,提出了基于圈复杂度的阶段动态符号执行 CCSDSEM 优化算法。该算法通过计算约束判定条件为真的数量来衡量代码的圈复杂度,然后依据圈复杂度的阈值进行分阶段动态符号执行,使动态符号执行梯度进行,缓解路径选择指数爆炸,合理缩放符号执行。最后在 KLEE 中实现 CCSDSEM 框架,并对测试程序做了检测。CCSDSEM 将符号执行的运行时间显著缩短,提升了生成测试用例的数量。

关键词:圈复杂度;动态符号执行;阶段执行;软件测试

中图分类号:TP393.08 **文献标识码**:A DOI:10.19358/j.issn.2096-5133.2020.04.005

引用格式:毕雪洁,於家伟,李世明.基于圈复杂度的阶段动态符号执行[J].信息技术与网络安全,2020,39(4):24-29.

Stage dynamic symbol execution based on cyclomatic complexity

Bi Xuejie[1], Yu Jiawei[1], Li Shiming[1,2]

(1. College of Computer Science and Information Engineering, Harbin Normal University, Harbin 150025, China;
2. Shanghai Key Laboratory of Integrated Administration Technologies for Information Security, Shanghai 200240, China)

Abstract: In order to alleviate problems such as the path explosion of dynamic symbol execution contraction, this paper proposes an optimization algorithm for stage dynamic symbol execution based on cyclomatic complexity (CCSDSEM). The algorithm realizes the cyclomatic complexity of the code by calculating the constraints to determine the exact number, and then executes the dynamic symbol execution in stages according to the threshold of cycle complexity, makes the dynamic symbol execution gradient, replaces the path selection index explosion, and scales the symbol execution reasonably. The above optimization framework was implemented in KLEE, and the test program was tested. CCSDSEM significantly changes the runtime of symbolic execution, increasing the number of test cases generated.

Key words: cyclomatic complexity; concolic execution; scaling symbolic execution; software testing

0 引言

路径爆炸问题降低了软件测试的效率和质量,也给软件埋下隐患。如何缓解路径爆炸问题成为软件安全测试中的一个研究热点,符号执行[1]成为缓解该问题严重程度的重要技术之一。其主要算法思想为利用符号变量来取代测试过程的真实用例,从而在执行过程中获取对应的执行路径,成为生成高覆盖测试用例和在复杂软件应用程序中查找深度错误的有效技术之一;因该技术能够处理复杂结构程序,开发人员也经常用之于程序自动测试[2]、程序缺陷检测[3]、测试用例生成[4]等。

*基金项目:上海市信息安全管理技术研究重点实验室开放课题(AGK2015003)

1 相关研究工作

1.1 符号执行

符号执行使用符号值而非真实用例作为输入值,并将程序变量的值表示为符号表达式[5]。因此,由程序计算的输出表示为符号输入的函数[6]。也就是说,每个符号执行的结果等同于大量测试用例,以便尽可能地检测程序中可能出现的行为和状态是否满足安全性能。符号执行技术分类主要包括以下几种。

(1)经典符号执行技术。主要特点为在理论上可以遍历更多的程序,但不能探索路径约束条件下可能无法处理的可行执行,故并未得到广泛应用。

(2)动态符号执行技术。主要有 Concolic 测试[7-8]和 EGT(Execution-Generated Testing)[9]等,能

够利用对实际输入得到的分支路径判定条件进行逻辑取反来探索所有可能的路径，如 EXE[10] 和 KLEE[11] 工具和扩展的 EGT 方法等。该技术在软件测试、逆向工程等应用方面得到认可。

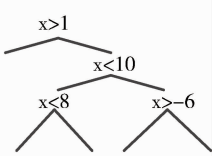
在动态符号执行时，外部代码的交互、约束求解（本文暂不过多讨论）因超时而降低精确性[1]，在使用真实用例缓解该问题时也破坏了路径遍历的完整性，其本质问题依然是路径爆炸和约束求解的难解，故其成为动态符号执行的技术瓶颈之一。优化路径选择可缓解上述问题，相关研究主要包括：(1)使用启发式搜索探求最佳路径，提高程序路径覆盖率，加快符号执行速度，包括：① 使用静态控制流图（CFG）选择路径；② 利用先决条件及输入特性进行符号执行，如预条件符号执行方法[4]、利用静态分析工具来分析程序中缺陷语句的方法[12]等。(2)利用程序分析技术减少路径探索的复杂性，包括：①构建函数和循环摘要供后续重用，如 Concolic 执行提出的可动态生成函数摘要组合方法[7]，可有效重新利用分析结果；② 剪枝冗余路径，如基于程序功能执行流切片技术[13]，通过裁剪掉与功能无关的分支路径来提高制导效率；③通过依赖性分析[14]来有效合并变量状态，提高路径分析的精度；④通过状态合并来缩小路径分支的选择范围，缓解路径爆炸问题，但该方法易降低路径覆盖率。

随着程序中逻辑判定语句的增加，路径爆炸问题越突出，符号执行面临的技术挑战越严峻[15]。

1.2 符号执行树

在符号执行过程中，根据执行路径而生成的树状结构表示定义为执行树，如图 1 所示。

```
I=<input>;
x=I-5;
if (x>1){
 if (x<10){
   if (x<8)
    ...
   }else {
    if (x>-6){
       ...
         }
    ...
    }
...
}
```



(a) 示例程序　　(b) 相应符号执行树

图 1　示例程序及生成的符号执行树

在执行树中，当程序执行到第 3 节点时，路径约束条件即为 $(x > 1) \wedge (x < 10) \wedge (x > -6)$，约束求解器（要解出 3 个约束条件的解，即算出可达路径的值）；此时若遍历该执行树，路径条数将以指数级数目增长并引发路径爆炸问题。

若执行树中存在一条由 n（当 $n \in Z^+$）个逻辑节点组成的路径，则程序执行时需对 n 个约束条件集合求解，当 $n \to +\infty$ 时其计算量无疑是巨大的；若此时对易引发路径爆炸的执行树进行分阶段符号执行，则可降低路径爆炸发生的概率和约束条件的求解难度，但须在分阶段符号执行前探测执行树的深度。

1.3 圈复杂度

圈复杂度（Cyclomatic Complexity）[16] 是用来衡量程序代码判定结构复杂程度的重要标准，其值越大说明代码的判断逻辑结构越复杂，即代码质量低或难于测试及维护，存在潜藏缺陷和漏洞的可能性就越大。

一般情况下，增加圈复杂度的核心问题实际是大量的逻辑判定语句，表现在代码上是 case、if、while 等语句及函数调用，而圈复杂度的计算有利于控制程序逻辑长度、设置检查点（或测试点）和评测代码质量。

定义 1　圈复杂度值 $V_{CC} = e - n + 2$，其中 e、n 分别表示代码对应控制流图中边数和节点数（含起点和终点，当有多个终点时只计算 1 次）。

定义 2　圈复杂度标准量级 G_{CC} 一般被定义为三个级别：Ⅰ级（代码质量优秀）、Ⅱ级（代码可重构或优化）和Ⅲ级（强制重构），如公式(1)所示。

$$G_{CC} = \begin{cases} \text{Ⅰ}, 0 \leqslant V_{CC} \leqslant 10 \\ \text{Ⅱ}, 11 \leqslant V_{CC} \leqslant 15 \\ \text{Ⅲ}, 16 \leqslant V_{CC} \leqslant +\infty \end{cases} \tag{1}$$

针对路径爆炸问题，本文提出了基于圈复杂度的阶段动态符号执行模型（Cyclomatic Complexity-based Stage Dynamic Symbolic Execution Model，CCS-DSEM），建立利用圈复杂度来探测圈复杂度高的代码段，然后以符号化逻辑判定分支数量作为衡量标准，分阶段进行动态符号执行操作，并在分段处进行约束集求解及优化，进而降低求解的复杂度。本文贡献如下。

(1)分段策略可降低程序逻辑判定分支的规

模,缓解路径爆炸现象。

(2)利用分段执行和优化约束条件求解,可提高符号执行的执行效率和路径覆盖率。

2 基于圈复杂度的阶段动态符号执行模型

2.1 模型定义

将待测代码作为模型外部输入数据并计算各检查点的圈复杂度及代码量,然后在统计结果的基础上根据求得的圈复杂度与阈值进行比较,来预判动态符号执行的计算量级,并据此设置检查点来分段执行对应的代码段,在运行符号执行引擎后分别生成缺陷报告。

定义 3:将基于圈复杂度的阶段动态符号执行模型定义为一个四元组 CCSDSEM = (I,P,E,O),其中:

(1) $I = (I_{SC_1}, I_{SC_2}, \cdots, I_{SC_i}, \cdots, I_{SC_n} \mid i,n \in Z^+)$ 表示该模型中被测序的源代码(Source Code,SC)。

(2) $P = (P_{CQ}, P_{LC}, P_{IE}, P_{CT}, P_{SSN})$ 表示对测试对象按一定策略进行系列处理;其中 P_{CQ} 表示统计代码量(Code Quantity,CQ),P_{LC} 表示统计循环条件(Loop Condition,LC),P_{IE} 表示按照约束规则进行信息提取(Information Extraction,IE),P_{CT} 表示计算阈值(Calculate the Threshold,CT),P_{SSN} 表示按逻辑结构进行阶段节点设置(Set Stage Node,SSN)。

(3) $E = (E_{SG}, E_{SEE})$ 表示对处理后的对象进行符号执行;其中 E_{SG} 表示符号生成器(Symbol Generator,SG),E_{SEE} 表示符号执行引擎(Symbol Execution Engine,SEE)。

(4) $O = (O_{DR_1}, O_{DR_2}, \cdots, O_{DR_i}, \cdots, O_{DR_n} \mid i,n \in Z^+)$ 表示输出的各个测试报告,其中 O_{DR_i} 表示输出的第 i 个缺陷报告(Defect Report,DR)。基于圈复杂度的阶段动态符号执行模型框架如图 2 所示。

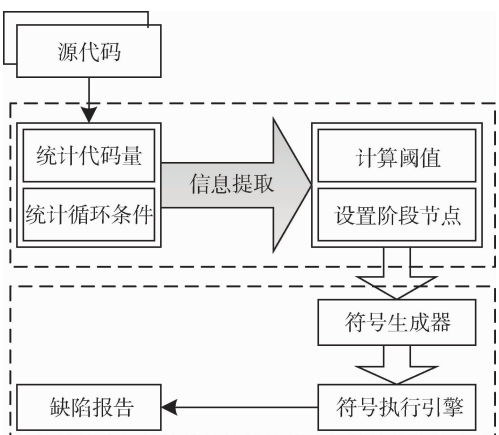


图 2 CCSDSEM 框架

2.2 分段执行规则

为便于阐述分段符号执行策略,现构造部分代码如下:

```
(1)  int x,y,z,t;
(2)  int t;
(3)  scanf("%d,%d,%d",&x,&y,&z);
(4)  if (x<1) t=x;
(5)        else if (x>10) t=2×x-1;
(6)          else if (x+y>10) t=2×y+x;
(7)            else if (z<0) t=x+z;
(8)              else if(y<12) t=y-z;
(9)                else t=fun(x);
(10) printf("%d\n",t);
(11) int fun(int n)
(12) {  int m;
(13)      m=2×fun(n-1)+1;
(14)      return m;  }
```

(1)Concolic 执行过程

在测试过程中,Concolic 执行会随机产生输入测试值(如{x = 3,y = 13,z = 5}),当执行 else if (x >10)时,符号化执行会生成路径约束($x_0 > 1$),然后 Concolic 执行对逻辑判断条件取反并对($x_0 < 1$)求解,并将得到的一个测试用例作为输入,从而执行不同路径。

同理,第 5 行产生约束条件($x_0 >= 1$) ∧ ($x_0 > 10$)以及($1 < x_0 <= 10$);重复执行上述过程,进而不断求出新的测试用例和探索路径。当执行到第 8 行时,约束条件为($1 <= x_0 < 10$) ∧ ($x_0 + y_0 > 10$) ∧ ($z_0 > 0$) ∧ ($y_0 < 12$)并对约束求解;随着程序执行的不断深入,约束条件复杂性变大,路径分支将会以指数级增长,加大了约束求解难度。

如果圈复杂度超过预先设定阈值 Φ 时,则对该部分代码进行分段执行或优化约束条件后再执行;此时,因¬($y_0 < 12$)是取反的分支约束条件,对($y_0 < 12$)分支无任何影响,故可去掉对 z 的约束,可从当前路径条件中移除与当前分支结果无关的约束,提高约束求解效率,降低因约束条件过于复杂而引发路径爆炸的概率。

(2)分阶段动态符号执行

当在 CCSDSEM 框架中执行分阶段动态符号执行时,根据控制流顺序设置检查点并计算对应代码段的圈复杂度,当某个圈复杂度大于预先设定的阈值 Φ 时,则对该段代码单独执行动态符号执行;反

复运用此策略，直至所有代码检测并分解完毕，算法描述如下：

```
算法：分阶段符号执行策略算法
输入：C 语言格式的待测代码
输出：高圈复杂度函数名称及复杂度值
(1) 初始化 CCSDSEM = (I,P,E,O)        //初始化模型
(2) 构建待测代码集 I
(3) int Φ = 50;                        //设定阈值 Φ = 50
                //遍历各函数并计算函数的圈复杂度 V_CC
(4) int CC(F_i);                       //计算 V_CC
(5)  { int i = 0;
(6)    while (I! = NULL)    //当输入集非空时执行
(7)    { i + +;                        //第 i 个函数
(8)    VCC = count (read (F_i));
                          //读取函数 F_i 并求出 V_CC
(9)    if (V_CC > Φ)                   //若 V_CC 大于 Φ
(10)   { output F_i, V_CC;             //输出 V_CC
(11)     temp_file_i = read(F_i);
                          //读取 F_i 代码到 temp_file_i
(12)     CC (temp_file_i); }  //递归计算 F_i 的 V_CC
(13)   else Symbolic_ Execution (F_i);
                                  //符号执行 F_i
(14)   }
(15)  }
```

当程序的整体影响处于可控状态下，采用此策略对被测程序按阈值进行分阶段动态符号执行，可降低产生路径爆炸的概率，缓解因动态符号执行将部分值实例化而引发的路径覆盖不足等问题。

3 实验与分析

本文实验通过 SourceMonitor 对开源项目 GNU Binutils-2.14 中部分源代码文件进行圈复杂度测试，选用 V_{CC} 值较高的源代码文件 readelf.c 进行测试。

3.1 圈复杂度计算实验

实验进一步对 readelf.c 计算圈复杂度，与本研究有关的详细信息如表 1 所示。

表 1 readelf.c 测试后的主要信息

序号	参数	值
1	源代码行数	10 694
2	语句条数	6 697
3	源代码功能数	7
4	最复杂的函数名	switch( )
5	最复杂的函数所占的行数	6 683
6	最复杂的函数的圈复杂度	517

在 readelf.c 中，以函数为检查点计算圈复杂度，可看出最复杂函数 switch( )（在分支语句控制下各函数名相同但内部代码实际是不同的，即各函数的 V_{CC} 值不同）的 V_{CC} 值非常高（值为 517），进一步计算得出各功能函数的详细信息，如表 2 所示。

表 2 readelf.c 测试后的主要信息

序号	函数名	圈复杂度	语句	最大深度
1	switch( )	517	1 612	32
2	process_file_header( )	16	48	5
3	VPARAMS( )	1	5	1
4	VPARAMS( )	1	5	1
5	usage( )	1	7	1
6	print_address( )	1	1	1
7	db_task_printsym( )	1	1	1

显然 switch( ) 为 readelf.c 的主要函数，而且明显高于其他函数，从其 Kiviat 图（如图 3 所示）上可以看出各项指标严重偏离合理区间（深色圆环区域）。

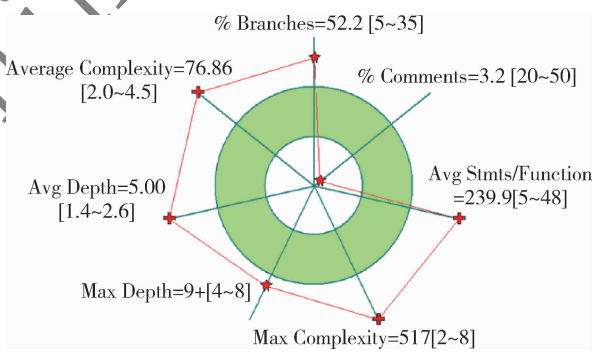


图 3 Kiviat 图

对所有 switch( ) 函数计算 V_{CC}，累计 97 个（其中 I 级 55 个，II 级 14 个，III 级 28 个），V_{CC} 最高为 95，最低为 2；在 III 级中 $V_{CC} > 50$ 的 switch(*) 如表 3 所示。

表 3 复杂度大于 50 的含参 switch( ) 函数

序号	函数名	圈复杂度
1	switch (attribute)	95
2	switch (e_machine)	93
3	switch (e_machine)	92
4	switch (attribute)	85
5	switch (type)	82
6	switch (type)	67
7	switch (tag)	65
8	switch (op)	61
9	switch (c)	61

3.2 分阶段动态符号执行实验

在对测试所得的 V_{CC} 实验数据分析后发现:若分支情况越多、嵌入层次越深、调用函数越多且复杂,则 V_{CC} 值越大。因符号执行在 V_{CC} 越大的情况下效率会严重降低,若将一个复杂程度分解为多个复杂度相对理想且内聚性强的代码片段来进行分阶段符号执行,显然具备缓解或降低路径爆炸的功效。

此部分实验选用符号执行工具 KLEE 2.0,CPU 为 Intel(R) Core(TM) i5-3320M 2.60 GHz、内存为 16 GB、Ubuntu 16.04.11 LTS、内核版本为 5.4.0-6,编译器 clang version 6.0.1,与 KLEE 相关如 llvm6.0,clang6.0。实验首先对将 $V_{CC} > \Phi$(Φ 设为 50)的 switch()函数调用采用剪枝处理(会影响代码调用,但会简化与其对应的执行树),V_{CC} 值越大该方法越有益。此外,实验中对于 case 语句非常多的情况,采用简单以 Φ 值为上限的简单分割方法,分割后代码段的 $V_{CC} \leqslant \Phi$;分阶段动态符号执行后的数据及原因分析如表 4 所示。

表 4 部分含参 switch()分阶段执行后的复杂度

序号	函数名	圈复杂度	代码特点及情况
1	switch (attribute)	12	调用 9 次 switch (uvalue)(V_{CC} 在 6-18 之间不等
2	switch (e_machine)	7	含有多条 case;又调用 switch (e_flags & EF_V850_ARCH)、switch ((e_flags & EF_MIPS_MACH))、switch ((e_flags & EF_MIPS_ABI))、switch ((e_flags & EF_MIPS_ARCH))、switch (e_flags & EF_PARISC_ARCH)
3	switch (e_machine)	7	此两个 switch (e_machine)执行树不同
4	switch (attribute)	50	全部为 case 语句
5	switch (type)	50	多数为 case 语句
6	switch (type)	50	调用 2 次 switch (elf_header.e_machine)
7	switch (tag)	50	全部为 case 语句
8	switch (op)	50	
9	switch (c)	50	多数为 case 语句,调用一次 switch (p[0])函数

3.3 局限性分析

(1)执行分阶段动态符号执行时,具体逻辑判断语句(如 case、if)及嵌套层次严重影响分阶段情况,相对应测试的圈复杂度之间差别也很大;尤其代码中 case 语句趋向全部时,最简单的分阶段方法为:当 case 语句数量等于 Φ 时进行分割(存在误判),此时 $V_{CC} = \Phi$(如表 4 中序号 4 ~ 9 情况)。如果循环或递归的终止条件是符号化的,就可能会出现无限数量的对应路径,因此对循环分支数量的判定结果是衡量符号执行树状状况的重要依据。

(2)实验情况可以判断出动态符号执行中可能产生路径爆炸的不同规模。目前这种方法并不能完全固定分支的数量,故每次实验时结果存在误差,但总体上可以满足圈复杂度不高于 Φ 的条件。

(3)本实验只对单个.C 程序文件进行测试,尚未对大型项目代码进行测试,也未考虑各文件之间的依赖关系等。

4 结论

本文首先介绍动态符号执行的相关研究现状和基本概念,然后详细分析存在的不足以及采用的分阶段执行方法,并分析路径覆盖率,进而提出基于圈复杂度的阶段动态符号执行的方法,并通过示例实验证明,相比于原有的方案,使用 CCCSDSEM 优化框架后降低了代码的圈复杂度并缓解了路径爆炸。

后续的研究工作主要从以下方面开展:(1)在 KLEE 分支状态跟踪中收集实际执行的路径数量并做出相应的判断及缓解路径爆炸的策略;(2)如何更好地结合圈复杂度的进行分段优化,并在 KLEE 中做相应改进是后续研究工作中的重点;(3)如何在分阶段动态符号执行过程中提高约束求解效率;(4)如何保证圈复杂度设置的值既不让程序分割过

多而消耗系统资源,又可一定程度上遍历更多的路径且缓解路径爆炸和约束求解。

参考文献

[1] CADAR C,SEN K. Symbolic execution for software testing: three decades later[J]. Communications of the ACM,2013,56(2):82-90.

[2] ZHANG J. Constraint solving and symbolic execution[C]. Working Conference on Verified Software: Theories, Tools, and Experiments. Springer, Berlin, Heidelberg, 2005: 539-544.

[3] BOYER R S,ELSPAS B,LEVITT K N. SELECT-a formal system for testing and debugging programs by symbolic execu-tion[J]. ACM SigPlan Notices,1975,10(6):234-245.

[4] AVGERINOS T,CHA S K,LIM B T H,et al. AEG: automatic exploit generation[C]. Proceedings of the Network and Distributed System Security Symposium (San Diego, CA, Feb. 6-9). Internet Society, Reston, VA, 2011, 283-300.

[5] KING J C. Symbolic execution and program testing[J]. Communications of the ACM,1976,19(7):385-394.

[6] CADAR C,GODEFROID P,KHURSHID S,et al. Symbolic execution for software testing in practice: preliminary assessment[C]. 2011 33rd International Conference on Software Engineering (ICSE). IEEE,2011:1066-1071.

[7] GODEFROID P,KLARLUND N,SEN K. DART: directed automated random testing[C]. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation,2005:213-223.

[8] SEN K,MARINOV D,AGHA G. CUTE: a concolic unit testing engine for C[J]. ACM SIGSOFT Software Engineering Notes,2005,30(5):263-272.

[9] CADAR C,ENGLER D. Execution generated test cases: how to make systems code crash itself[C]. International SPIN Workshop on Model Checking of Software. Springer, Berlin,Heidelberg,2005:2-23.

[10] CADAR C,GANESH V,PAWLOWSKI P M,et al. EXE: automatically generating inputs of death[J]. ACM Transac-tions on Information and System Security (TISSEC),2008,12(2):1-38.

[11] CADAR C,DUNBAR D,ENGLER D R. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs[C]. OSDI,2008,8:209-224.

[12] 崔展齐,王林章,李宣东. 一种目标制导的混合执行测试方法[J]. 计算机学报,2011,34(6):953-964.

[13] 甘水滔,王林章,谢向辉,等. 一种基于程序功能标签切片的制导符号执行分析方法[J]. 软件学报,2019,30(11):3259-3280.

[14] 郭曦,王盼. 基于变量符号关联分析的程序状态优化方法[J]. 通信学报,2018,39(6):81-88.

[15] 张健,张超,玄跻峰,等. 程序分析研究进展[J]. 软件学报,2019,30(1):80-109.

[16] EBERT C,CAIN J. Cyclomatic complexity[J]. IEEE software,2016,33(6):27-29.

(收稿日期:2020-03-04)

作者简介:

毕雪洁(1994 -),女,硕士研究生,主要研究方向:网络与信息安全、漏洞挖掘。

於家伟(1994 -),男,硕士研究生,主要研究方向:网络与信息安全、漏洞挖掘。

李世明(1976 -),通信作者,男,硕士,副教授,主要研究方向:网络与信息安全、物联网技术、数据挖掘。E-mail:hsdlsm@163.com。

版权声明